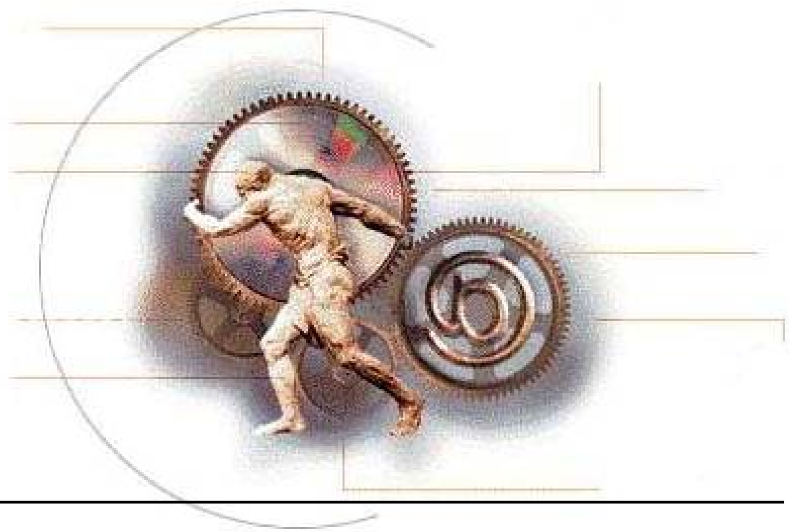


Course 6

動態規劃
Dynamic Programming

國立聯合大學 資訊管理學系 陳士杰老師

Outlines

- 本章重點
  - Divide-and-Conquer v.s. Dynamic Programming
  - Dynamic Programming v.s. Greedy Approach
  - Floyd's Algorithm for Shortest Paths
  - Chained Matrix Multiplication
  - Dynamic Programming and Optimization Problems

Divide-and-Conquer v.s. Dynamic Programming

- 由前一單元得知，Divided-and-Conquer即為遞迴解法。
- 以費氏數 (Fibonacci Number) 說明:

終止條件

$$F_n = \begin{cases} 0, & if \quad n = 0 \\ 1, & if \quad n = 1 \\ F_{n-1} + F_{n-2}, & if \quad n \geq 2 \end{cases}$$

遞迴關係

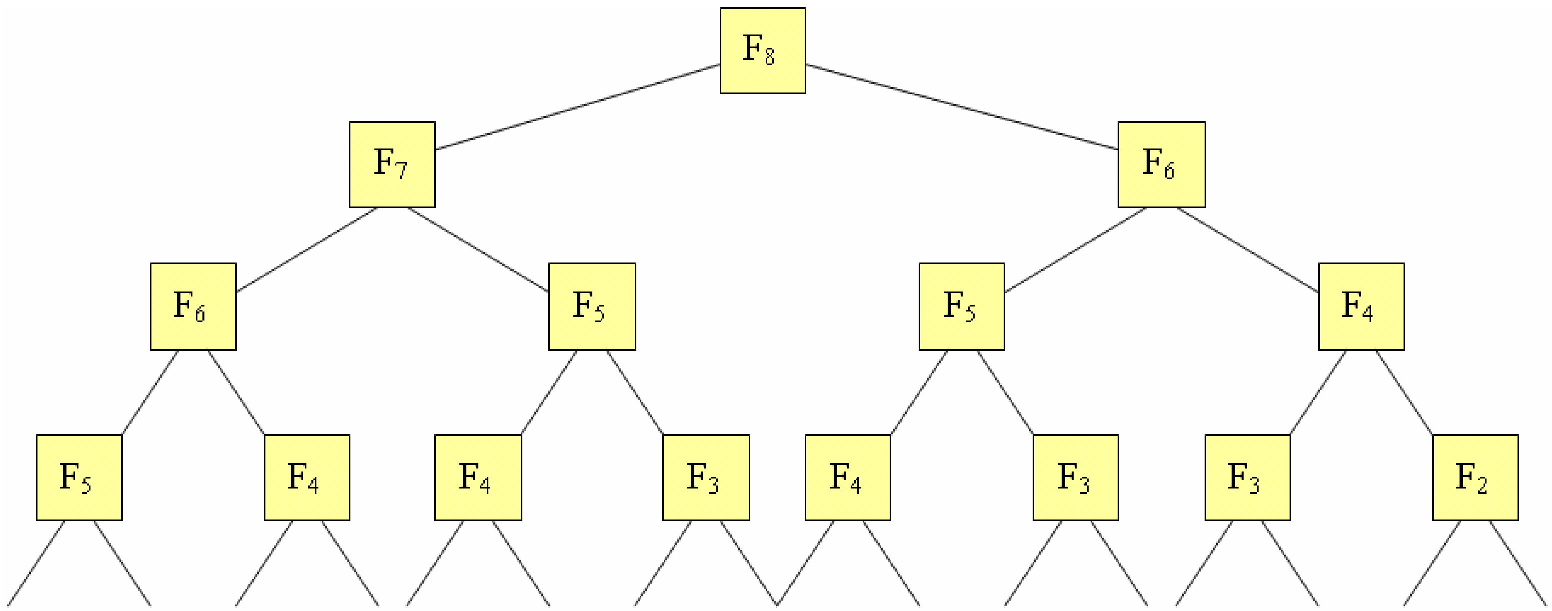
Recursive Fibonacci Algorithm

inputs: num identified the ordinal of the Fibonacci number

outputs: returns the nth Fibonacci number

```
void Fib(int n)
{
    if (n is 0 OR n is 1)
        return n;
    else
        return (Fib(n-1) + Fib(n-2));
}
```

- **Based on recursive function, 求取Fib (8)的過程如下:**



- **Top-Down求算方式**
- **子問題重覆 (Overlapping Subproblem)**
  - ➪是**Divided-and-Conquer**主要的問題所在

- 以人的方式求算:

n	0	1	2	3	4	5	6	7	8
F_n	0	1	1	2	3	5	8	13	21

- 求算F_8和F_7時，F_6會被用到2次，但我們用表格記錄已算過的部份!!
- Bottom-Up求算方式
- 動態規劃 (Dynamic Programming)是一種表格式的演算法設計原則。
  - 其精神是將一個較大的問題定義爲較小的子問題組合，先處理較小的問題，並將其用表格儲存起來，再進一步地以較小問題的解逐步建構出較大問題的解。
  - Programming: 用表格存起來，有 "以空間換取時間"之涵意
    - 屬於作業研究 (OR)的技巧

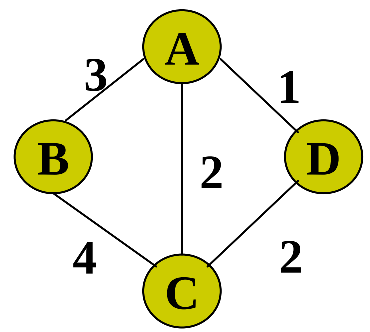
- Divide-and-Conquer 和 Dynamic Programming都是將問題切割再採用遞迴方式處理子問題，但是:
  - Divide-and-Conquer可能會對相同子問題進行**重覆計算**
  - Dynamic Programming會使用**表格**將子問題的計算結果加以儲存，在後面階段如果需要這個計算結果，再直接由表格中取出使用，因此可以避免許多重覆的計算，以提高效率。
- Divide-and-Conquer v.s. Dynamic Programming

	Divide-and-Conquer	Dynamic Programming
額外記憶體空間	不需額外記憶體空間	需要額外記憶體空間
解題方式	Top-Down	Bottom-Up
適用時機	適用non-overlap子問題	適用overlap子問題

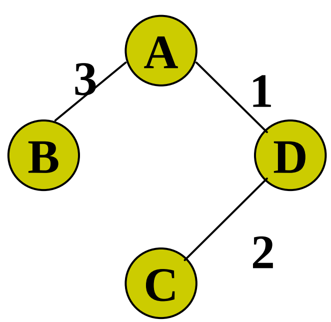
Dynamic Programming v.s. Greedy Approach

- 對於**具有限制的最佳化問題**，可以採用 “貪婪法則” 或 “動態規劃” 來設計演算法則。
  - Greedy Approach:
    - 是一種**階段性 (Stage)** 的方法
    - 具有一**選擇程序 (Selection Procedure)**，自某起始點(值) 開始，在每一個階段逐一檢查每一個輸入是否適合加入答案中，重複經過多個階段後，即可順利獲得最佳解
    - **較為簡單** (∴若遇最佳化問題，先思考可否用Greedy Approach解，若不行再考慮用Dynamic Programming)
    - 如果所要處理的最佳化問題<u>無法找到一個選擇程序來逐一檢查</u>，則需要以<u>一次考慮所有的可能情況</u>的方式來處理，就是屬於Dynamic Programming
  - Dynamic Programming
    - 先把<u>所有的情況都看過一遍</u>，才去挑出最佳的結果
    - 考慮問題所有可能的情況，將最佳化問題的目標函數表示成一個遞迴關係式，結合Table的使用以找出最佳解

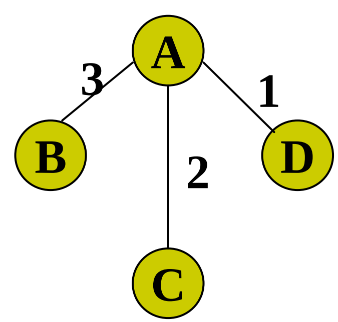
- **Ex 1.** 有一Graph如下，每一個邊都有一權重值，試找出 "具最小權重值總合且不包含Cycle" 的 Graph。



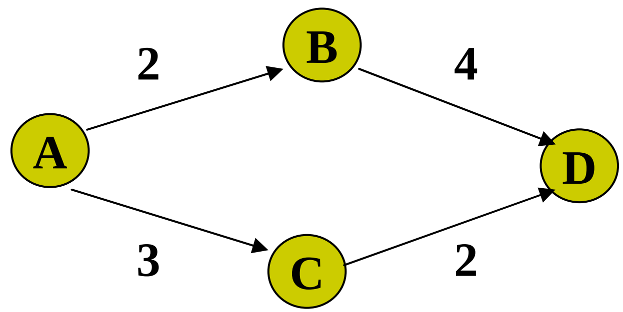
- **Sol:** (選擇程序) 從最小的邊開始逐一選擇，挑選出來的邊不能構成Cycle，直到所有的邊都被選完爲止。



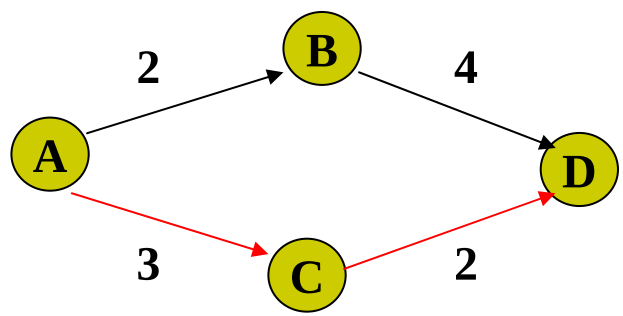
或



- **Ex 2.** 一有向加權圖Graph如下，該圖可分成2個部份，請找出由第一個部份的出發頂點到最後部份的目的地頂點的最短距離路。



- **Sol:** 找不到一個選擇程序，可自某起始點(值) 開始逐一檢查每一個輸入是否適合加入答案中。

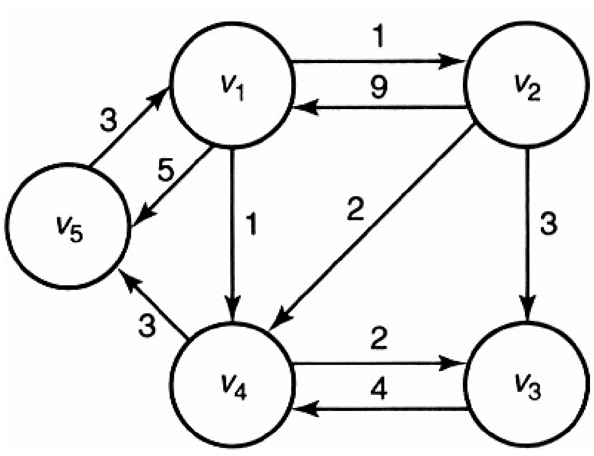


Dynamic Programming的解題方式

- 使用Dynamic Programming解決問題的過程如下：
  - 先找出問題的**遞迴式** (遞迴關係式; Recurrence Relations)
  - 接著，透過Forward Approach或是Backward Approach，利用**迴圈**與**表格**來處理遞迴式，進而解決問題。
    - 假設有n個數值 ($X_1, X_2, \ldots, X_{i-2}, X_{i-1}, \boxed{X_i}, X_{i+1}, X_{i+2}, \ldots, X_{n-1}, X_n$)
    - **Forward Approach** (前向法；順向法)
      - 如果要計算出X_i的值，需要透過前方($X_{i+1}, \ldots, X_{n-1}, X_n$)等數值資料
      - 由X_n向後求解問題的解答(Solved Backward)
    - **Backward Approach** (後向法；逆向法)
      - 如果要計算出X_i的值，需要透過後方($X_1, X_2, \ldots, X_{i-1}$)等數值資料
      - 由X_1向前求解問題的解答(Solved Forwards)

Floyd's Algorithm for Shortest Paths

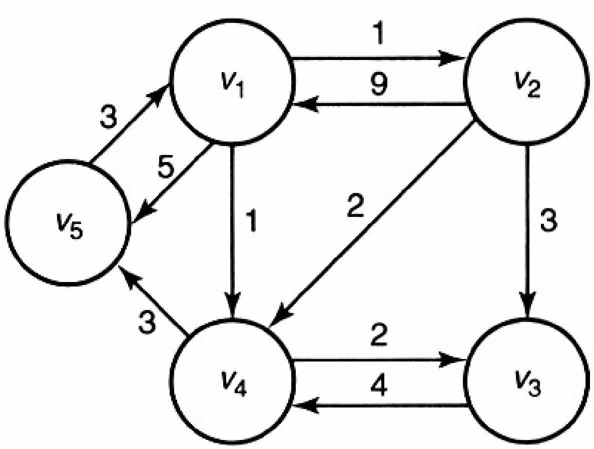
- **最短路徑 (Shortest Path) 問題**
  - **求單一頂點到其它頂點之最短路徑 (Single pair shortest path)**
    - 使用**Dijkstra's Algorithm**
      - 採用 "貪婪演算法" 之解題策略
      - 找出某一頂點到其它頂點之最短路徑之時間複雜度為$O(n^2)$
  - **求所有頂點之間的最短路徑 (All pair shortest path)**
    - 使用**n次Dijkstra's Algorithm**
      - 每一次帶不同的起始點
      - 需要的時間複雜度 $O(n^3)$
    - 使用**Floyd's Algorithm**
      - 採用 "動態規劃" 之解題策略



- 最短路徑問題其實是**最佳化問題(Optimization problem)**。
  - 一個最佳化問題可能會有一個以上的**候選解 (Candidate Solution)**。
  - 每一個候選解都具有一個**值 (Value)**，而該問題的解就是具有最佳值的那個解。
  - 隨著問題的不同，最佳解可能是要求最小值，也可能是最大值。
- 因此，在Shortest Paths problem中：
  - 一個***Candidate solution***就是從某個頂點到另一個頂點的路徑
  - 候選解的***value***則是該條路徑的長度
  - ***Optimal value***則是這些長度當中最小的值

- **All-pair Shortest Path問題**
  - 給定一個有向加權圖形 (Directed Weighted Graph)，G=(V, E)，找出任意兩個頂點 ($v1, v2 \in V$) 之間的最短路徑
- 在Figure 3.2中，v_1 到 v_3存在三條簡單路徑：$[v_1, v_2, v_3]$，$[v_1, v_4, v_3]$ 與 $[v_1, v_2, v_4, v_3]$ 。由於：

簡單路徑(Simple path)：該路徑中，同一頂點不會出現兩次。

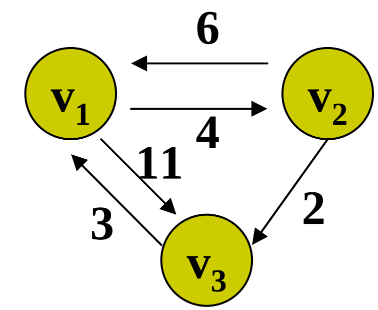


$$length\,[v_1, v_2, v_3] = 1 + 3 = 4$$
$$length\,[v_1, v_4, v_3] = 1 + 2 = 3$$
$$length\,[v_1, v_2, v_4, v_3] = 1 + 2 + 2 = 5,$$

因此，$[v_1, v_4, v_3]$ 是 v_1 到 v_3的最短路徑。

- 佛洛依德最短路徑演算法 (Floyd's Algorithm for Shortest Paths) :
  - Floyd-Warshall Algorithm
  - 假設G=(V, E)，|V| = n
  - **D^k矩陣**: 為一 n×n的矩陣，其中$D^k[i, j]$表示自 v_i 至 v_j ($v_i \rightarrow v_j$)的最短路徑長，且途中經過的頂點編號均 **≤ k** (其中k≥0)

- 範例:



$D^1[2, 1] = 6$

2 → 1
2 → 1 → 2 → 1
2 → 3 → 1

$D^2[2, 1] = 6$

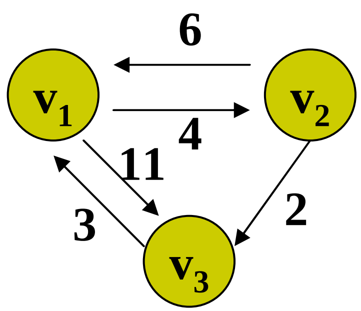
2 → 1
2 → 1 → 2 → 1
2 → 3 → 1

$D^3[2, 1] = 5$

2 → 1
2 → 1 → 2 → 1
2 → 3 → 1

∵節點個數為 3
∴ $D^3[i, j]$可得到總體最短路徑

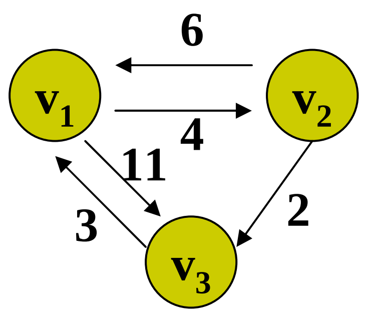
- 當k = 0時， 矩陣$D^0[i, j]$表示為**Adjacency Matrix (相鄰矩陣; W)**。
  - 自 v_i 至 v_j 途中不會經過其它頂點



$$D^0 = W = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} \begin{array}{ccc} v_1 & v_2 & v_3 \end{array} \\ \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix} \end{array}$$

- Floyd's Algorithm求解過程:
  - 找出相鄰矩陣 W
  - 逐步求出$D^1, D^2, \ldots, D^n$矩陣
  - D^n矩陣即為結果

- 求解右圖的**All-pair Shortest Path**



Sol:

- 找出相鄰矩陣 W

$$D^0 = W = \begin{array}{c c} & \begin{array}{ccc} v_1 & v_2 & v_3 \end{array} \\ \begin{array}{c} v_1 \\ v_2 \\ v_3 \end{array} & \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix} \end{array}$$

- 逐步求出D¹, D², D³矩陣
  - **Step 1:** 由 W 矩陣求出 D¹ 矩陣

$$W = \begin{array}{c c} & \begin{array}{ccc} v_1 & v_2 & v_3 \end{array} \\ \begin{array}{c} v_1 \\ v_2 \\ v_3 \end{array} & \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix} \end{array} \Rightarrow D^1 = \begin{array}{c c} & \begin{array}{ccc} v_1 & v_2 & v_3 \end{array} \\ \begin{array}{c} v_1 \\ v_2 \\ v_3 \end{array} & \begin{bmatrix} \mathbf{0} & \mathbf{4} & \mathbf{11} \\ \mathbf{6} & \mathbf{0} & \mathbf{2} \\ \mathbf{3} & \mathbf{7} & \mathbf{0} \end{bmatrix} \end{array}$$

$$\begin{cases} 2 \to 3 = 2 \\ 2 \to 1 \to 3 = 17 \end{cases}$$

$$\begin{cases} 3 \to 2 = \infty \\ 3 \to 1 \to 2 = 7 \end{cases}$$

- **Step 2:** 由 D¹ 矩陣求出 D² 矩陣

$$\mathbf{D}^1 = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} \begin{array}{ccc} v_1 & v_2 & v_3 \end{array} \\ \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array} \Rightarrow \mathbf{D}^2 = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} \begin{array}{ccc} v_1 & v_2 & v_3 \end{array} \\ \begin{bmatrix} \mathbf{0} & \mathbf{4} & \mathbf{6} \\ \mathbf{6} & \mathbf{0} & \mathbf{2} \\ \mathbf{3} & \mathbf{7} & \mathbf{0} \end{bmatrix} \end{array}$$

$$\begin{cases} 1 \to 3 = 11 \\ 1 \to 2 \to 3 = 6 \end{cases}$$

$$\begin{cases} 3 \to 1 = 3 \\ 3 \to 2 \to 1 = \infty \end{cases}$$

- **Step 3:** 由 D² 矩陣求出 D³ 矩陣

$$\mathbf{D}^2 = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} \begin{array}{ccc} v_1 & v_2 & v_3 \end{array} \\ \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array} \Rightarrow \mathbf{D}^3 = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} \begin{array}{ccc} v_1 & v_2 & v_3 \end{array} \\ \begin{bmatrix} \mathbf{0} & \mathbf{4} & \mathbf{6} \\ \mathbf{5} & \mathbf{0} & \mathbf{2} \\ \mathbf{3} & \mathbf{7} & \mathbf{0} \end{bmatrix} \end{array}$$

$$\begin{cases} 1 \to 2 = 4 \\ 1 \to 3 \to 2 = \infty \end{cases}$$

$$\begin{cases} 2 \to 1 = 6 \\ 2 \to 3 \to 1 = 5 \end{cases}$$

Algorithm 3.3: Floyd's Algorithm for Shortest Paths

Problem: Compute the shortest paths from each vertex in a weighted graph to each of the other vertices. The weights are nonnegative numbers.

Inputs: A weighted, directed graph and *n*, the number of vertices in the graph. The graph is represented by a two-dimensional array *W* which has both its rows and columns indexed from 1 to *n*, where *W*[*i*][*j*] is the weight on the edge from the *i*th vertex to the *j*th vertex.

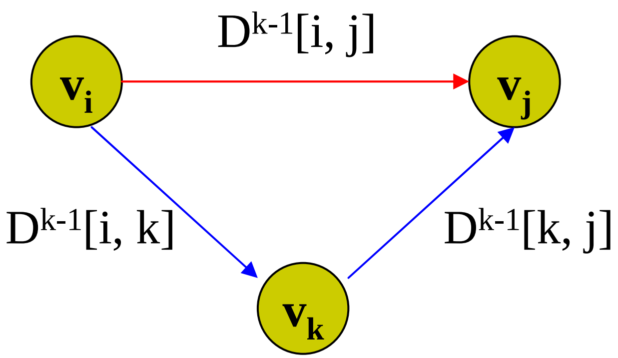
Outputs: A two-dimensional array *D*, which has both its rows and columns indexed from 1 to *n*, where *D*[*i*] [*j*] is the length of a shortest path from the *i*th vertex to the *j*th vertex.

```
void floyd (int n
                    const number W[] []
                    number D[] []
{
 index i, j, k;
 D = W;
 for (k = 1; k<=n; k++)   //k爲 D^k 的k值
     for (i = 1; i <= n; i++)
         for (i = 1; j <= n; j++)
           D[i][j] = minimum(D[i][j], D[i][k] + D[k][j]);
}
```

Time Complexity: $O(n^3)$

◆ Floyd's Algorithm觀念圖解:

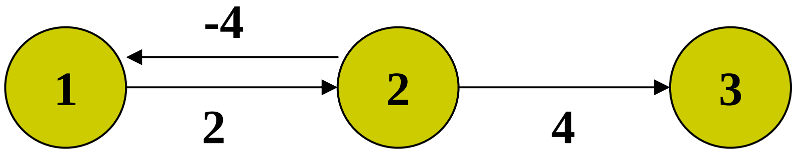
- 求矩陣$D^k[i, j]$，是由矩陣$D^{k-1}[i, j]$而來



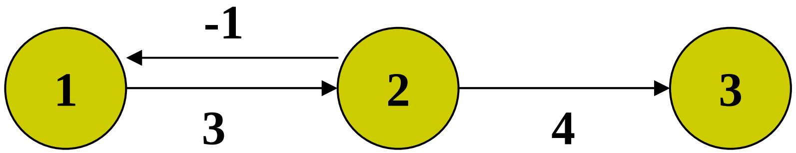
- 遞迴式：

$$D^k[i,j]=\begin{cases} W[i,j], & \text{if } k=0 \\ \min(D^{k-1}[i,j], D^{k-1}[i,k]+D^{k-1}[k,j]), & \text{if } k>0 \end{cases}$$

- Floyd's Algorithm的假設條件:
  - 圖形中不得有負長度的Cycle存在
  - 例 1: 1 →2的最短路徑為 __



  - 例 2: 1 →2的最短路徑為 _



■連鎖矩陣相乘(Chained Matrix Multiplication)

- 假設我們要將一個 **2 × 3** 的矩陣乘上一個 **3 × 4** 的矩陣：

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 29 & 35 & 41 & 38 \\ 74 & 89 & 104 & 83 \end{bmatrix}$$

  - 所產生的結果爲一個 **2 × 4**的矩陣
  - 結果矩陣中的每一個元素都必須經過**3**次乘法的運算
  - 因爲結果矩陣中有 **2 × 4 = 8**個元素，因此總共需要的乘法次數爲：

$$2 \times 4 \times 3 = 24.$$

- 一般來說，一個 ***i* × *j* matrix** 乘上一個 ***j* × *k* matrix** ，總共需要的乘法次數爲：

$$i \times j \times k \text{ elementary multiplications.}$$

- **Note: n個matrix相乘有** $C_{n-1} = \binom{2(n-1)}{n-1} \Big/ n$ 種可能的配對組合(括號方式)
  - **Ex:** 以下有四個矩陣相乘:

$$\begin{array}{ccccccc} A & \times & B & \times & C & \times & D \\ 20 \times 2 & & 2 \times 30 & & 30 \times 12 & & 12 \times 8 \end{array}$$

由**Note**得知共有五種不同的相乘順序，不同的順序需要不同的乘法次數：

$$\begin{array}{ll} A(B(CD)) & 30 \times 12 \times 8 + 2 \times 30 \times 8 + 20 \times 2 \times 8 = 3,680 \\ (AB)(CD) & 20 \times 2 \times 30 + 30 \times 12 \times 8 + 20 \times 30 \times 8 = 8,880 \\ A((BC)D) & 2 \times 30 \times 12 + 2 \times 12 \times 8 + 20 \times 2 \times 8 = 1,232 \\ ((AB)C)D & 20 \times 2 \times 30 + 20 \times 30 \times 12 + 20 \times 12 \times 8 = 10,320 \\ (A(BC))D & 2 \times 30 \times 12 + 20 \times 2 \times 12 + 20 \times 12 \times 8 = 3,120 \end{array}$$

其中，以第三組是最佳的矩陣相乘順序。

- **Chained Matrix Multiplication:**
  - Def: 給一Matrix Chain: $A_1, A_2, \ldots, A_n$，求此Chain所需之乘法次數爲最少之括號方式 (即: 最佳的矩陣乘法組合方式)。
- 若$A_i, A_{i+1}, \ldots, A_j$ 在某組合方式所需的乘法次數爲最小 (最佳)，則必存在一個k，使得$A_i, A_{i+1}, \ldots, A_k$ 和$A_{k+1}, A_{k+2}, \ldots, A_j$皆爲最佳。

最佳組合

$((A_i\ A_{i+1} \ldots A_k)(A_{k+1}\ A_{k+2} \ldots, A_j))$

最佳子組合 最佳子組合

- Matrix Chain的遞迴式

$$M[i,j]=\begin{cases}0 & \text{if } i=j\\ \min\limits_{i\le k\le j-1}\{M[i,k]+M[k+1,j]+d_{i-1}d_k d_j\} & \text{if } i<j\end{cases}$$

- 此遞迴式涵蓋以下兩個規則：
  - M[i][j] = 矩陣Ai乘到Aj所需的最少乘法數 (其中 i < j)
  - M[i][i] = 0

- **Chained Matrix Multiplication** 問題的演算法需有兩個表格和一個主要變數:
  - **M[i, j]**
    - 記錄多個矩陣相乘 **(e.g., A_i × … × A_j)**時，所需的 "最少" 乘法次數
  - **P[i, j]**
    - 記錄多個矩陣相乘 **(e.g., A_i × … × A_j)** 所需最少乘法次數之 "最佳乘法順序" 是由哪一個矩陣開始分割
  - **diagonal**
    - 主要指出在**Matrix Chain**中，每一次有多少個矩陣要相乘
      - **diagonal = 1** ⇨ 只有**1**個矩陣，∴不會執行乘法動作
      - **diagonal = 2** ⇨ 每一次有**2**個矩陣要相乘
      - **diagonal = 3** ⇨ 每一次有**3**個矩陣要相乘
      - **diagonal = 4** ⇨ 每一次有**4**個矩陣要相乘
      - **…**

Example 3.5

Suppose we have the following six matrices:

$$\begin{array}{ccccccccccc} A_1 & \times & A_2 & \times & A_3 & \times & A_4 & \times & A_5 & \times & A_6 \\ 5\times 2 & & 2\times 3 & & 3\times 4 & & 4\times 6 & & 6\times 7 & & 7\times 8 \\ d_0 \quad d_1 & & d_1 \quad d_2 & & d_2 \quad d_3 & & d_3 \quad d_4 & & d_4 \quad d_5 & & d_5 \quad d_6 \end{array}$$

To multiply A_4, A_5, and A_6, we have the following two orders and numbers of elementary multiplications:

$$\begin{aligned} (A_4A_5)\,A_6 \text{ Number of multiplications} &= d_3\times d_4\times d_5 + d_3\times d_5\times d_6 \\ &= 4\times 6\times 7 + 4\times 7\times 8 = 392 \\ A_4\,(A_5A_6) \text{ Number of multiplications} &= d_4\times d_5\times d_6 + d_3\times d_4\times d_6 \\ &= 6\times 7\times 8 + 4\times 6\times 8 = 528 \end{aligned}$$

Therefore,

$$M[4][6] = minimum(392, 528) = 392.$$

- 六個矩陣相乘的最佳乘法順序可以分解成以下的其中一種型式：

  1. $A_1 (A_2A_3A_4A_5A_6)$
  2. $(A_1A_2) (A_3A_4A_5A_6)$
  3. $(A_1A_2A_3) (A_4A_5A_6)$
  4. $(A_1A_2A_3A_4) (A_5A_6)$
  5. $(A_1A_2A_3A_4A_5) (A_6)$

- 第**k**個分解型式所需的乘法總數，爲前後兩部份 **(**一爲**A_1, A_2, ..., A_k** 和**A_{k+1}, ..., A_6)** 各自所需乘法數目的最小值相加，再加上相乘這前後兩部份矩陣所需的乘法數目。

$$M[1][6] = \underset{1 \le k \le 5}{minimum}(M[1][k] + M[k+1][6] + d_0 d_k d_6).$$

- **Matrix Chain的遞迴式**

$$M[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k \le j-1} \{M[i,k] + M[k+1,j] + d_{i-1}d_k d_j\} & \text{if } i < j \end{cases}$$

- **Example: $A^1_{3\times3}$, $A^2_{3\times7}$, $A^3_{7\times2}$, $A^4_{2\times9}$, $A^5_{9\times4}$,** 求此五矩陣的最小乘法次數。

Sol:

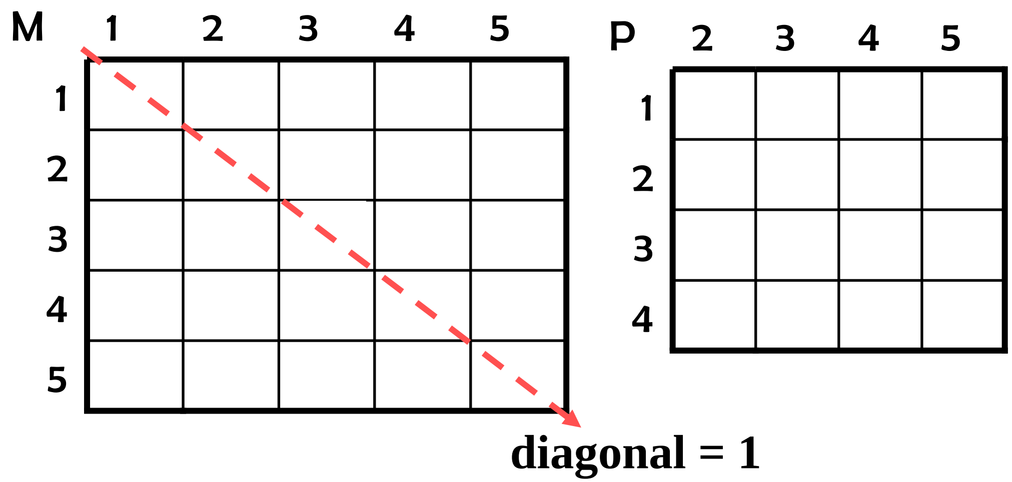
建立兩陣列 **M[1…5, 1…5]**及**P[1…4, 2…5]**

M	1	2	3	4	5
1					
2					
3					
4					
5					

P	2	3	4	5
1				
2				
3				
4				

Case ① (When diagonal = 1)

- diagonal = 1，∵只有1個矩陣，∴不會執行乘法動作
- 陣列M的中間對角線為0，陣列P則不填任何數值



Case ② (When diagonal > 1)

- diagonal = 2，有2個矩陣相乘
- 當 i = 1及 j = 2，為A^1及A^2矩陣相乘，此時:

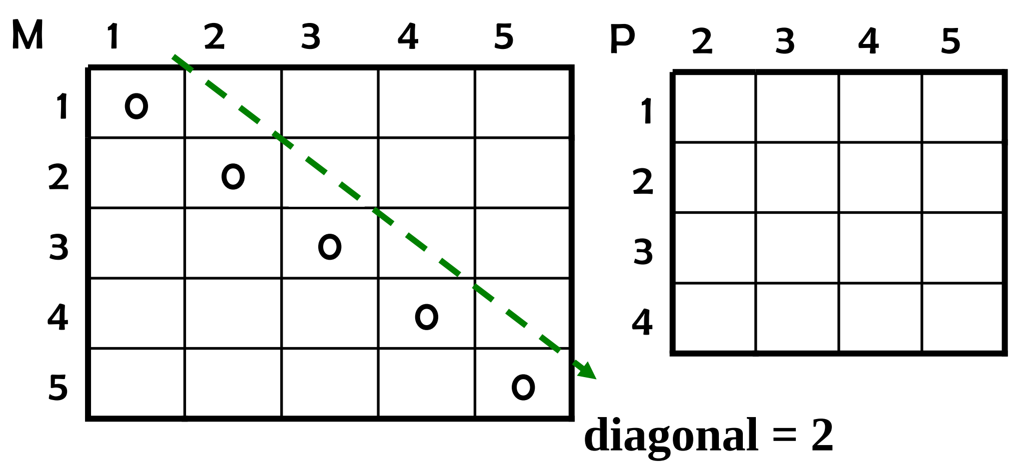
M[1, 2] = M[1,1]+M[2,2]+3×3×7 = 63，

其中 A^1 及A^2 的分割點 k 如下:

$A^1 \times A^2$

↑

分割點k = 1



Case ② (When diagonal > 1)

- diagonal = 3，有3個矩陣相乘
- 當 i = 2及 j = i+diagnal-1 = 2+3-1=4，為A^2至A^4間的所有矩陣相乘，此時:

$$M[2,4] = \min\begin{cases} M[2,2]+M[3,4]+3\times7\times9=315, & \text{分割點}k=2 \\ M[2,3]+M[4,4]+3\times2\times9=96, & \text{分割點}k=3 \end{cases}$$

M	1	2	3	4	5
1	0	63	60		
2		0	42	96	
3			0	126	128
4				0	72
5					0

diagonal = 3

P	2	3	4	5
1	1	1		
2		2	3	
3			3	3
4				4

Case ② (When diagonal > 1)

- diagonal = 4，有4個矩陣
- 當 i = 1及 j = 4，為A^1至A^4間的所有矩陣相乘，此時:

M	1	2	3	4	5
1	0	63	60	114	
2		0	42	96	138
3			0	126	128
4				0	72
5					0

diagonal = 4

P	2	3	4	5
1	1	1	3	
2		2	3	3
3			3	3
4				4

$$M[1,4] = \min\begin{cases} M[1,1] + M[2,4] + 3\times 3\times 9 = 177, & \text{分割點}k = 1 \\ M[1,2] + M[3,4] + 3\times 7\times 9 = 378, & \text{分割點}k = 2 \\ M[1,3] + M[4,4] + 3\times 2\times 9 = 114, & \text{分割點}k = 3 \end{cases}$$

Case ② (When diagonal > 1)

- diagonal = 5，有5個矩陣
- 當 i = 1及 j = 5，為A^1至A^5間所有矩陣相乘，此時:

M	1	2	3	4	5
1	0	63	60	114	156
2		0	42	96	138
3			0	126	128
4				0	72
5					0

diagonal = 5

P	2	3	4	5
1	1	1	3	3
2		2	3	3
3			3	3
4				4

$$M[1,5] = \min\begin{cases} M[1,1] + M[2,5] + 3\times3\times4 = 174, & \text{分割點}k = 1 \\ M[1,2] + M[3,5] + 3\times7\times4 = 275, & \text{分割點}k = 2 \\ M[1,3] + M[4,5] + 3\times2\times4 = 156, & \text{分割點}k = 3 \\ M[1,4] + M[5,5] + 3\times9\times4 = 222, & \text{分割點}k = 4 \end{cases}$$

- **[Note]**此演算法的概念如下:

A^1, A^2, A^3, A^4, A^5 diagonal = 1

A^1A^2, A^2A^3, A^3A^4, A^4A^5 diagonal = 2

$A^1A^2A^3$, $A^2A^3A^4$, $A^3A^4A^5$ diagonal = 3

$A^1A^2A^3A^4$, $A^2A^3A^4A^5$ diagonal = 4

$A^1A^2A^3A^4A^5$ diagonal = 5

```
for (i = 1; i < = n; i++)
        M[i] [i] = 0;
for (diagonal = 2; diagonal < = n; diagonal ++)
    for (i = 1; i <= n – diagonal+1; i++) {
        j = i + diagonal+1;
        M[i] [j] = minimum (M[i] [k] + M[k + 1] [j] + d[i - 1]* d[k] * d[j]);
                   i ≤ k ≤ j - 1
        P[i] [j] = a value of k that gave the minimum;
    }
return M[1] [n];
```

Case ① (When diagonal > 1)

Case ② (When diagonal > 1)

```
for (i = 1; i < = n; i++)
        M[i] [i] = 0;
for (diagonal = 2; diagonal < = n; diagonal ++)
      for (i = 1; i <= n – diagonal+1; i++) {
          j = i + diagonal+1;
          M[i,j] = ∞;
          for (k = i; k <= j-1; k++)
         {
          q = M[i] [k] + M[k + 1] [j] + d[i - 1]* d[k] * d[j];
          if q < M[i] [j]
               {
                M[i,j]=q;
                P[i] [j] = k;
               }
          }
    }
return M[1] [n];
```

Dynamic Programming and Optimization Problems

- Dynamic Programming 和 Greedy Approach看似可以處理所有的最佳化問題，然而它們所能處理的最佳化問題需滿足下列原則
  - 最佳化原則(Principle of Optimality): 當一個問題存在著最佳解，則表示其所有子問題也必存在著最佳解
    - 以最短路徑問題來看，如果v_k是 v_i 到 v_j間最短路徑上的一個頂點，則 v_i 到 v_k 以及從 v_k 到 v_j 這兩個子路徑也必定是最短路徑.

- 每一個輸入的計算都必須是根據先前輸入的最佳結果再進一步計算，如此才能夠得到最佳解，同時可以將無法獲得最佳解的情況去除，以避免需要將每一種可能情況都加以考慮。
  - 對於n個輸入的最佳化問題 (X1, X2, …, Xn):
    - 有些被歸類為 “部份集合” (Subset) 問題，會有2^n種可能的情況
    - 有些被歸類為 “排列組合” (Permutation) 問題，會有n!種可能的情況
  - 因此皆屬指數複雜度的問題，若可採用最佳化原則通常可以將這一些問題的複雜度由指數複雜度降為多項式複雜度。
- 然而，並不是所有求最佳化的問題都合乎最佳化原則，此時就只能用其它的方法求解了。

Example 3.4

Consider the Longest Paths problem of finding the longest simple paths from each vertex to all other vertices. We restrict the Longest Paths problem to simple paths because with a cycle we can always create an arbitrarily long path by reapeatedly passing through the cycle. In Figure 3.6 the optimal (longest) simple path form v_1 to v_4 is $[v_1, v_3, v_2, v_4]$. However, the subpath $[v_1, v_3]$ is not an optimal (longest) path from v_1 to v_3 because

$$length\,[v_1, v_3] = 1 \qquad \text{and} \qquad length\,[v_1, v_2, v_3] = 4.$$

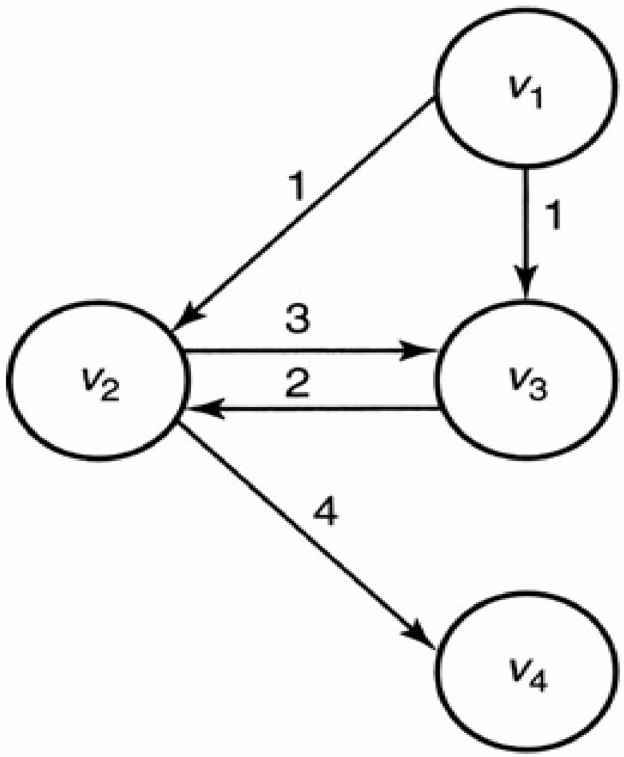


Figure 3.6: A weighted, directed graph with a cycle.

Therefore, the principle of optimality does not apply. The reason for this is that the optimal paths from v_1 to v_3 and from v_3 to v_4 cannot be strung together to given an optimal path from v_1 to v_4. Doing this would create a cycle rather than an optimal path.

補 充

補 1. The Longest Common Subsequence (LCS) Problem

- 假設有一條字串(String) S:
  - S = "a t g a t g c a a t"
  - Substrings of S: "g a t g c", "t g c a a t".
  - Subsequences of S: "a g g t", "a a a a".
- **String**
  - a segment of *consecutive* characters.
  - usually called *sequence* in Biology.
- **Sequence**
  - need not be consecutive.
- In Biology, **String = Sequence**.

- Common subsequences of two string X = < B, C, B, A, C > and Y = < B, D, A, B, C > :
  - BC, BA, BB, BAC, BBC, …
  - The **longest common subsequence (LCS)** of X and Y = BAC or BBC
- [Note]
  - LCS可能不唯一
  - 若用暴力法來解，必須先產生字串X的所有子序列，再逐一檢查這些子序列是否存在於字串Y中。如果字串X的長度爲m，則X會產生 2^m個子序列。因此，採用暴力法來解共同子序列問題的時間複雜度會是指數等級，不適用於長的序列。

- 求LCS 問題：
  - 給定兩個Sequence $X=\langle x_1, x_2, \ldots, x_m\rangle$和$Y=\langle y_1, y_2, \ldots, y_n\rangle$，求X和Y所構成的最長共同子序列$LCS(X, Y) = Z = \langle z_1, z_2, \ldots, z_k\rangle$為何。
- **LCS問題的遞迴式**
  - 令c[i, j]表示為兩個序列 $X = \langle x_1, x_2, \ldots, x_i\rangle$ 和 $Y = \langle y_1, y_2, \ldots, y_j\rangle$ 所構成之最長共同子序列的長度，則：

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{且} x_i = y_j \\ Max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{且} x_i \neq y_j \end{cases}$$

- **LCS問題的遞迴式設計概念：**
  - 若 i 或 j 為 0，表示 X 或 Y 這兩條序列的其中一條為空序列。
  - 若 $x_i = y_j$，則 c[i, j]所表示的序列長度，是由<x_1, x_2, …, x_{i-1}>和<y_1, y_2, …, y_{j-1}>兩序列所構成之最長共同子序列的長度(即: c[i-1, j-1])再加上 1。
  - 若 $x_i \neq y_j$，則 c[i, j]的序列長度，是由下列兩個不同的最長共同子序列長度當中之最大值所構成：
    - <x_1, x_2, …, x_{i-1}> 和 <y_1, y_2, …, y_j> 兩序列所構成之最長共同子序列的長度 c[i-1, j]
    - <x_1, x_2, …, x_i> 和 <y_1, y_2, …, y_{j-1}> 兩序列所構成之最長共同子序列的長度 c[i, j-1]

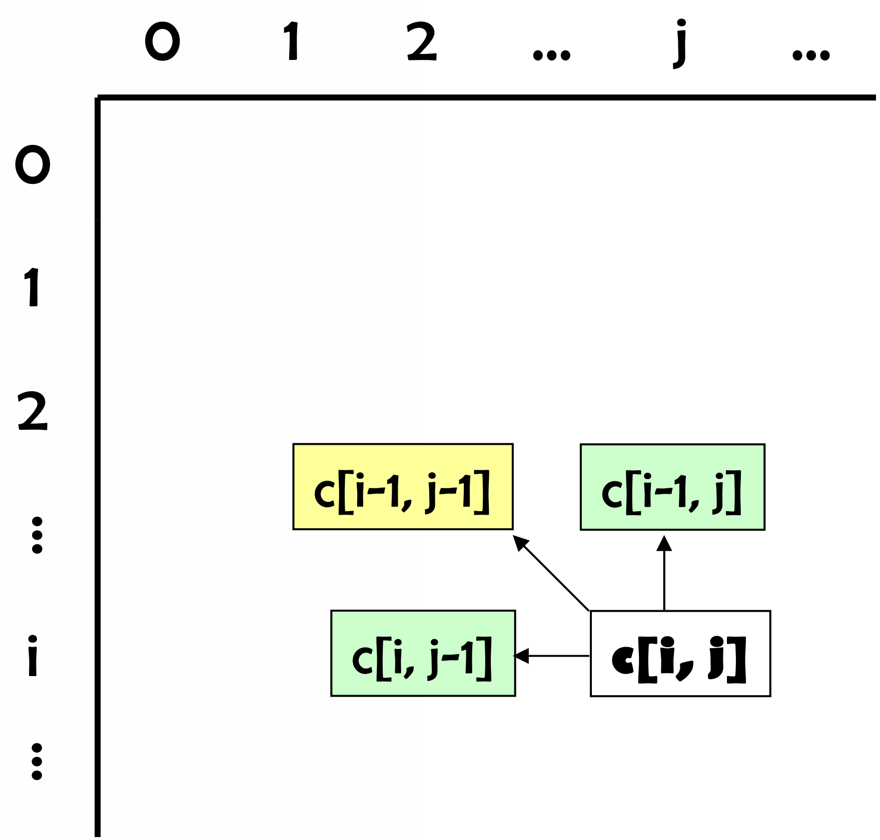
範例說明

- **Ex. 1 (當 $x_i = y_j$):**
  - **X = <A, B, C, D>　Y = <A, C, D>**
  - 共同子序列：**<A>, <C>, <D>, <A, C>, <A, D>, <C, D>, <A, C, D>**
  - 最長共同子序列 **Z = <A, C, D>**
    - 長度為**3**，此序列長度是由**<A, B, C>**和**<A, C>**所構成之最長共同子序列的長度再加上**1**
- **Ex. 2 (當 $x_i \neq y_j$):**
  - **X = <A, B, C, D>　Y = <D, B, C>**
  - 共同子序列：**<D>, <B>, <C>, <B, C>**
  - 最長共同子序列 **Z = <B, C>**
    - 長度為**2**，序列長度是由下列兩個不同的最長共同子序列長度當中之最大值所構成：
      - **<A, B, C>** 和 **<D, B, C>** 兩序列所構成之最長共同子序列的長度: **2**
      - **<A, B, C, D>** 和 **<D, B>** 兩序列所構成之最長共同子序列的長度: **1**

Inputs: 兩個 Sequence $X=\langle x_1, x_2, \ldots, x_m\rangle$和 $Y=\langle y_1, y_2, \ldots, y_n\rangle$
Outputs: LCS(X, Y)及其長度 c[m, n]

```
int c[0…m, 0…n];    //m 為序列 X 的長度；n 為序列 Y 的長度
index label[0…m, 0…n];
for (i=0; i ≤ m; i++)
      c[i, 0] = 0;
for (j=0; j ≤ n; j++)
      c[0, j] = 0;
for (i=1; i≤m; i++)
{
  for (j=1; j≤n; j++)
  {
      if (xi = yj) then
          {
            c[i, j] = c[i-1, j-1] +1;
            label[i, j] = "↖";
          }
      else if (c[i-1, j] ≥ c[i, j-1]) then
               {
                 c[i, j] = c[i-1, j];
                 label[i, j] = "↑";
               }
            else
               {
                 c[i, j] = c[i, j-1];
                 label[i, j] = "←";
               }
  }
}
```

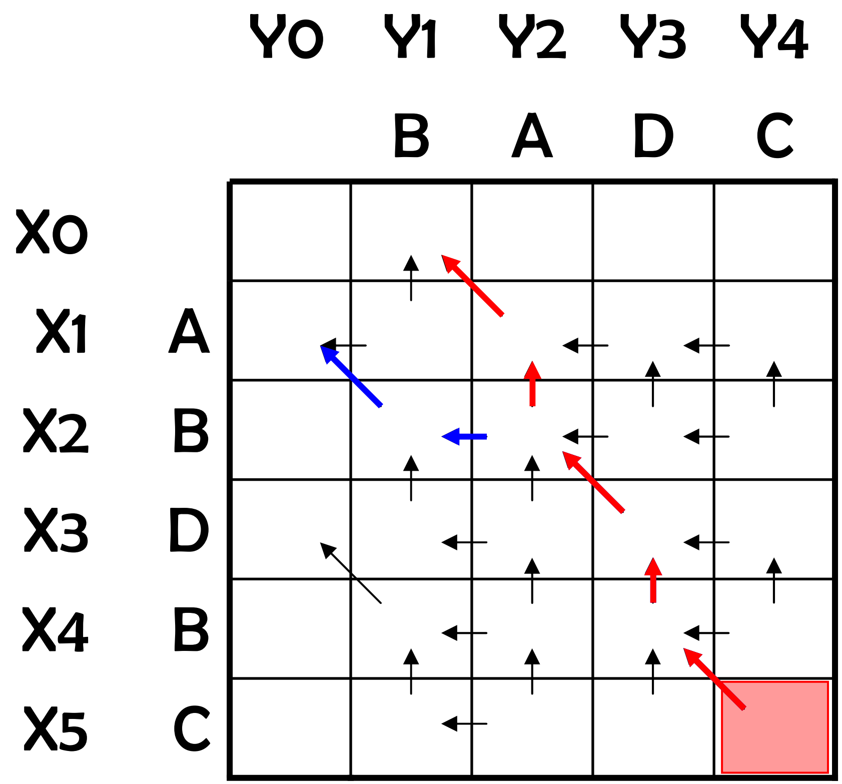
Time Complexity: O(mn)
Space Complexity: O(mn)



- Give two sequences X = <A, B, D, B, C> and Y = <B, A, D, C>. Find the longest common subsequence of X and Y.

Ans:

- 最長共同子序列的長度 = 3
- 最長共同子序列：
  - <A, D, C>
  - <B, D, C>



補2. The Sequence Alignment

- Compare two or more sequences
- Some reasons to perform the sequence alignment operations
  - Measure the similarity of some sequences.
  - A DNA sequence X and a database containing a set of DNA sequences.
  - Data compression.

- We have two DNA sequences S_1 and S_2:

Better

S_1	G	A	A	C	T	G			
S_2	G	A	G	C	T	G			

2 +2 -1 +2 +2 +2

Score=9

S_1	G	A	A	C	T	G	---	---	---
S_2	G	A	---	---	---	G	C	T	G

2 +2 -1 -1 -1 +2 -1 -1 -1

Score=0

- A scoring rule to measure the goodness of an alignment:
  - ai=bj, Score=2
  - ai or bj align with a blank, Score=-1
  - ai≠bj, Score=-1

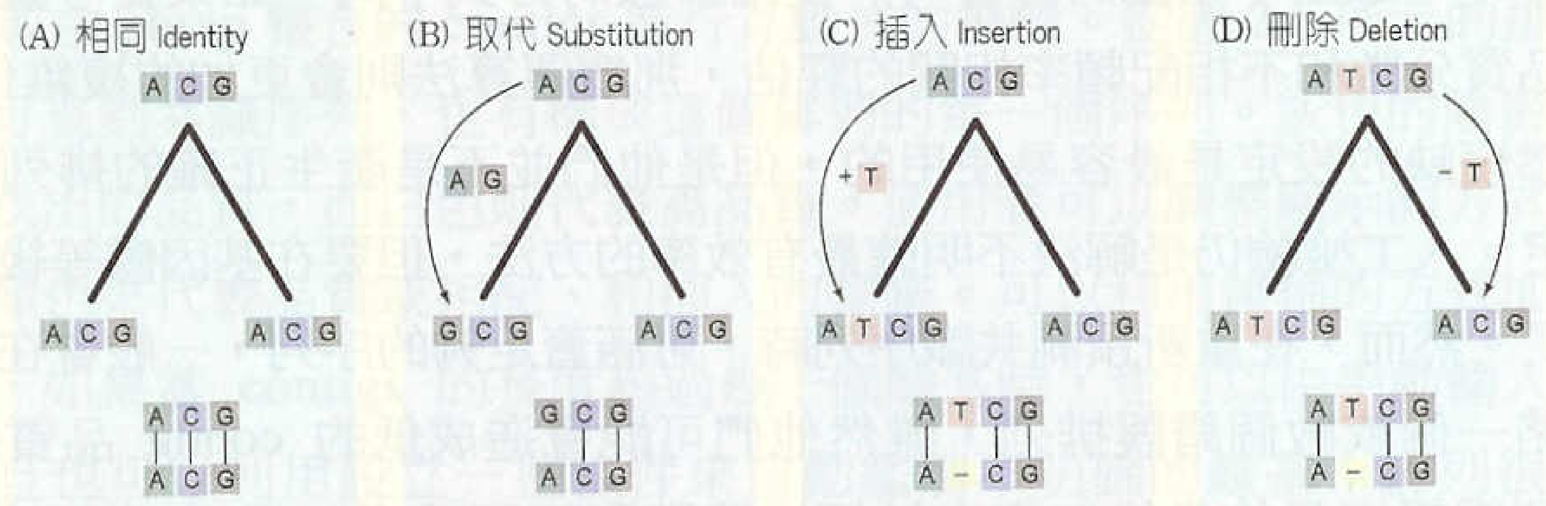
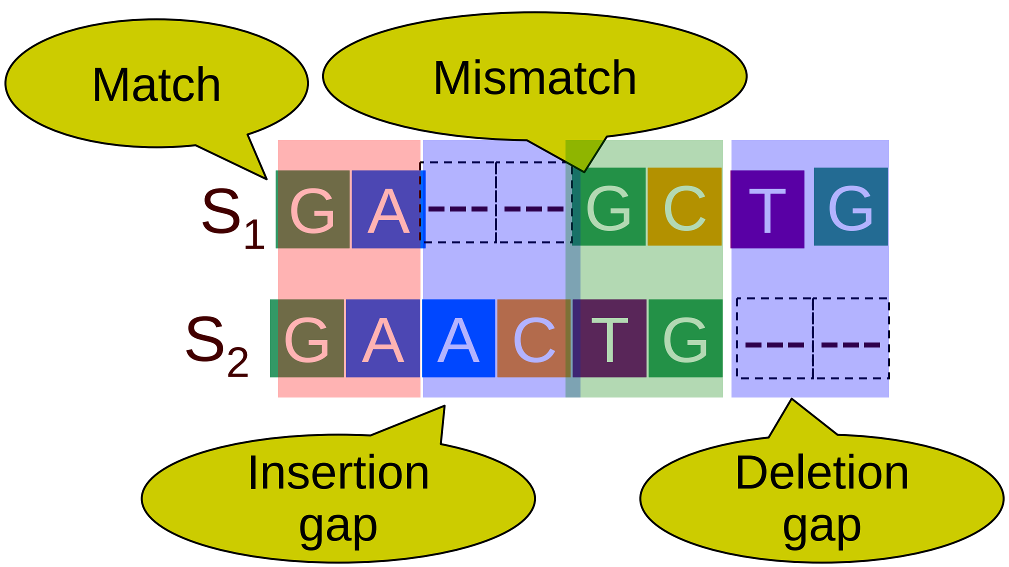


圖 Legend　一般的演化事件與它們在序列排列。在上圖中表示了序列可能的發生情形，在圖 A 中，序列裡可能有兩個位置是相同的；圖 B 中顯示序列中可能有一個以上的替代作用；或是在圖 C 與圖 D 中，由於插入或者缺失所造成序列上的缺口或是所謂 indel 的情形。

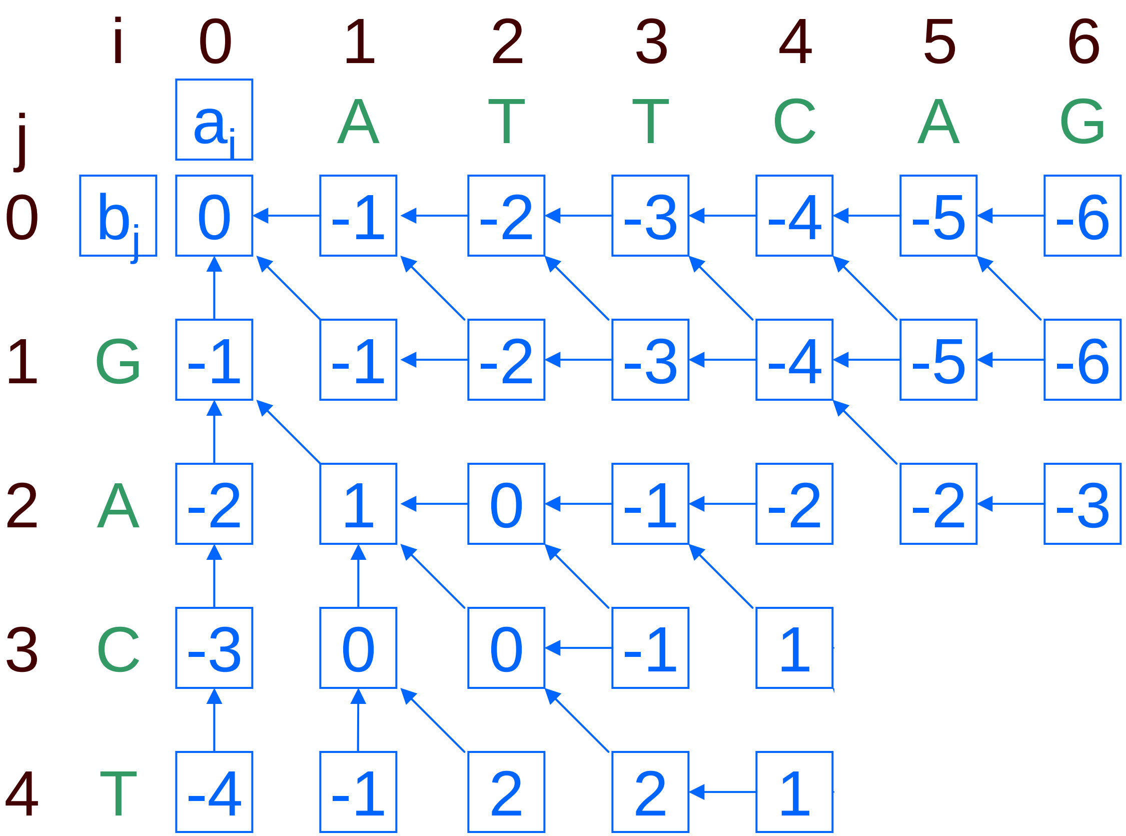
Sequence A: GAACTG
Sequence B: GAGCTG

An alignment of A and B:



- Find an alignment which has the highest score.

S_1: A T T C A G S_2: G A C T



A(i,j)=the score of optimal alignment

A(0,0)=0

A(i,0)= -i

A(0,j)= -j

If $a_i = b_j$, then

A(i,j)= A(i-1,j-1) +2

Else

A(i,j)=Max(A(i-1,j) –1,
A(i,j-1) –1,
A(i-1,j-1) –1)

S_1 A T T C A G S_2 G A C T

-	A	T	T	C	A	G
G	A	C	T	-	-	-

-	A	T	T	C	A	G
G	A	-	-	C	T	-

-	A	T	T	C	A	G
G	A	-	-	C	-	T

j \ i		0	1 A	2 T	3 T	4 C	5 A	6 G
0		0	-1	-2	-3	-4	-5	-6
1	G	-1	-1	-2	-3	-4	-5	-6
2	A	-2	1	0	-1	-2	-2	-3
3	C	-3	0	0	-1	1	0	-1
4	T	-4	-1	2	2	1	0	-1

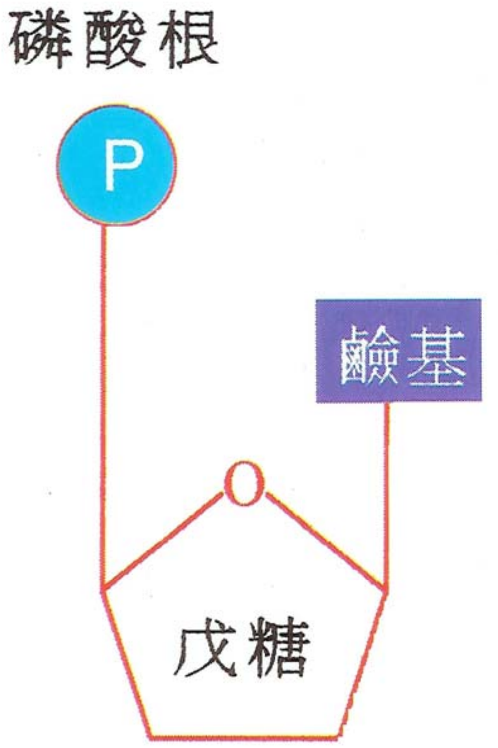
Tracing back the table

補 3. 核酸 (Nucleic acids)

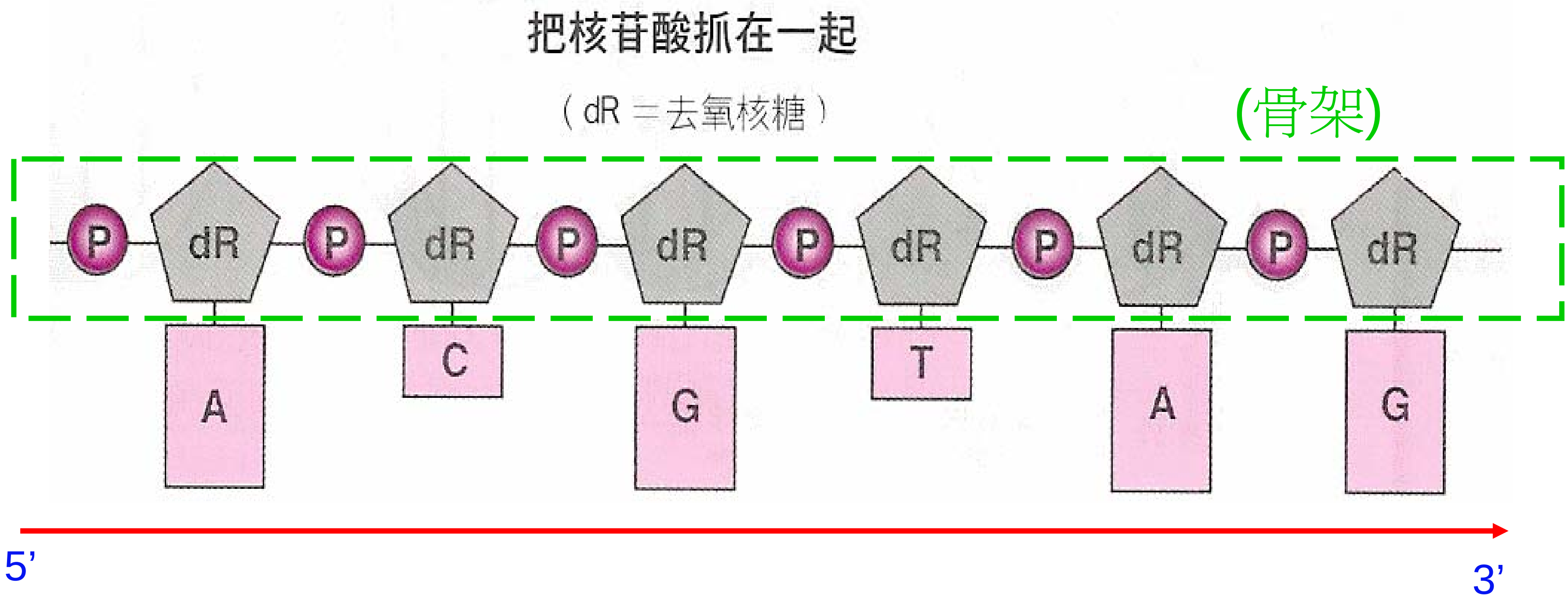
- 核酸是以核苷酸 (Nucleotide)爲單元體所聚成的巨分子，乃細胞內分子量最鉅大的功能性分子，包括 ：
  - 去氧核糖核酸 (Deoxyribonucleic acid，DNA)
  - 核糖核酸 (Ribonucleic acid，RNA)
- 核酸主要功能爲遺傳訊息的貯存、傳遞 與 表現，是現代分子生物學的主角。
- 由於DNA與RNA是由許多個核苷酸連接而成；因此，我們可以得知核酸是聚合物，單體是核苷酸。

核苷酸 (Nucleotide)

- 核苷酸的概要結構：
  - 磷酸根 (phosphate group)
  - 戊糖 (pentose sugar；又稱五碳糖) -可以是去氧核糖 (deoxyribose) 或者是核糖 (ribose)，造成 DNA 與 RNA 的差別。
  - 鹼基 (base；又稱含氮鹼基(nitrogenous base))
- DNA是由A、T、G、C四個不同的鹼基組成。
- RNA是由A、U、G、C四個不同的鹼基組成。



核苷酸的概要結構


把核苷酸抓在一起
(dR = 去氧核糖)
(骨架)
P
dR
A
C
G
T
A
G
5'
3'

補 4. Longest Increasing Subsequence (LIS)

- 給一數字序列 X = <x_1, …, x_m>，找出X之最長遞增子序列。
  - Ex: X = <5, 1, 4, 2, 3>，則 LIS(X) = <1, 2, 3>
- Algorithm:
  ① Y ← Sort(X); //Y具有遞增性
  ② Z ← LCS(X, Y); //找出X與Y之間的最長共同子序列，且具有遞增性。
  ③ Return Z

❖範例❖

- X = <5, 1, 4, 2, 3>，找出 LIS(X)。

<Ans>:

① Y ← Sort(X)。∴Y = <1, 2, 3, 4, 5>

② Z ← LCS(X, Y);

Z = <1, 2, 3>, LIS長度為3

③ Return Z

		Y0	Y1	Y2	Y3	Y4	Y5
			1	2	3	4	5
X0		0	0	0	0	0	0
X1	5	0	0	0	0	0	1
X2	1	0	1	1	1	1	1
X3	4	0	1	1	1	2	2
X4	2	0	1	2	2	2	2
X5	3	0	1	2	3	3	3